Data mining di Dati Scientifici

Manuel Jerez González

Project in Computer Science and Engineering

—

Academic Year 2021/2022

—

Politecnico Milano

1. Introduction

The aim of this report is to explain the problem, context and solutions proposed during my work for the 2021/2022 edition of the course "Project in Computer Science and Engineering" from Politecnico Milano. During this course, I had to select a topic between a list of different options, each of them related to different topics. I decided to take the project named "Data mining di dati scientifici" tutored by Professor Edoardo Ramalli.

The work done is framed in the SciExpeM project, powered by the Electronics, Information and Bioengineering Department (DEIB) from Politecnico Milano.

2. Problem

3. Context

SciExpeM is a project started by DEIB in collaboration with the chemical engineering department from Politecnico Milano. The goal of the project is to develop and maintain predictive models on combustion kinetics, that helps in the future to improve the efficiency, increase sustainability, reduce chemical pollutants sent to the atmosphere and help in the research and development of green fuels. This is done thanks to the big amount of experimental data available on this topic, and also thanks to the high computational resources on nowadays PCs that allow the project to use machine learning to push forward on the topic.

The studies on the combustion kinetics have been carried on and its data has been collected since the end of the WWII, and they continue today. Of course, the methods of acquiring the experimental data and the form of representation have been changing along the time, which leads to some uncertainties and the necessity to work on this. For my topic of research, the focus is on the kinetic models, which describe the different elements that appear in the chemical reactions that happen during the combustion.

As it has been said, to achieve the ultimate goal of SciExpeM, we need to be able to understand the data we are using, as there is a huge number of different ways to express the information we need to manage. One of the things we need to observe is the different possible kinetics models that are available where the chemical components are defined and inserted into the system.

The work that I was requested to do follows this topic, which is, given different chemical kinetics models, whose formats will be explained later, be able to compare them, disambiguating the different elements of the files with all the data and, with this information, conclude if species and reactions given are the same even if the chemical models and names used are different.

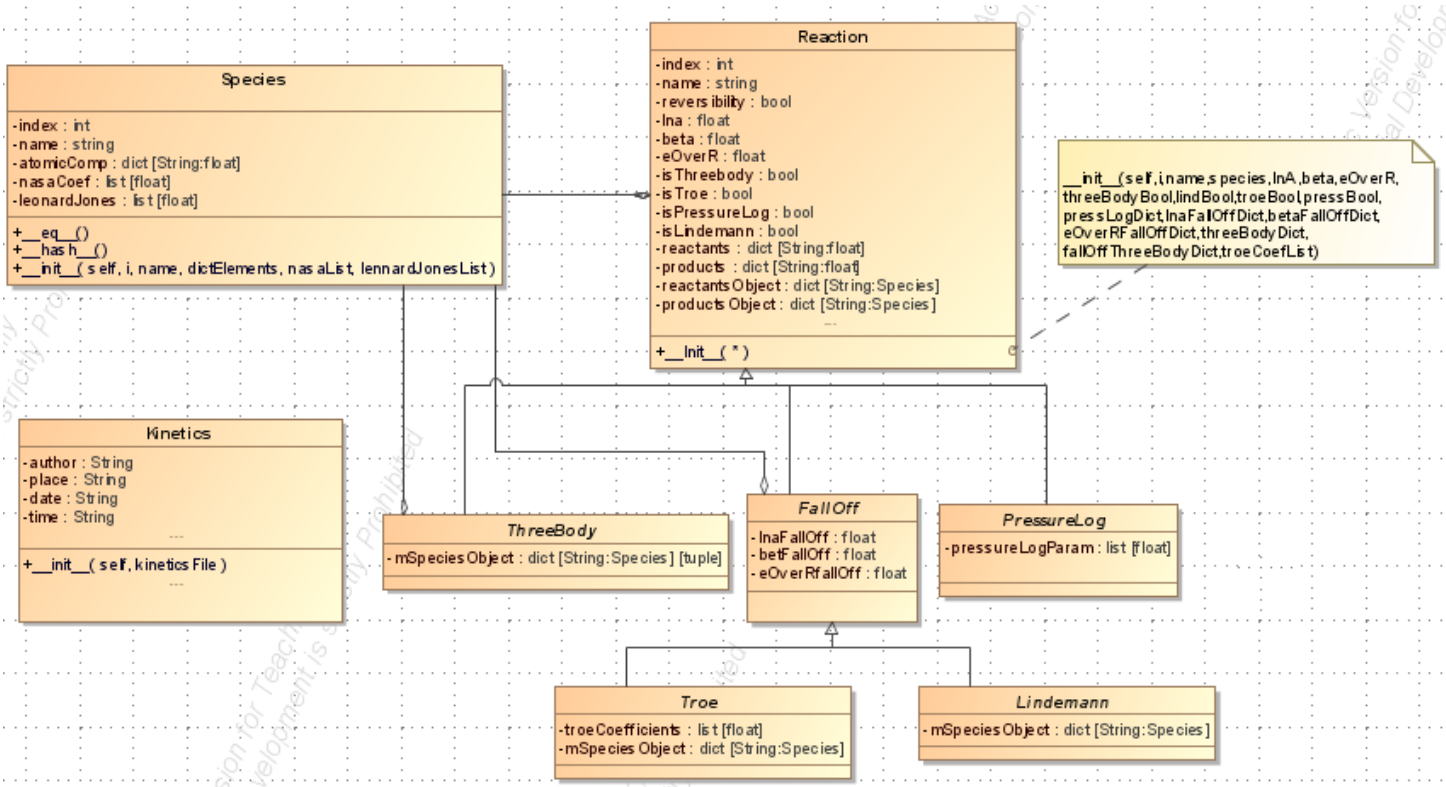
To do this, nevertheless, we need to attend other topics, such as the acquisition of data from specific files, the creation of models on Python to store the information, and stablish ways to compare two species coming from different kinetic models.

The approach that we followed to organize my work was to have meetings with my tutor every 2/3 weeks on average, discuss the work previously done and set the tasks that I had to do for the upcoming weeks.

4. Solutions

4.1. Creating Python Models

The first thing that was to be done to start working on the project was to understand the type of data that we had on the files, so we could decide how many classes, and which attributes had to be created. Then, once this was understood, we could start working on the Python model. Before coding, I decided to make UML diagrams, to make everything more clear and also to make easier the latter works on the code. Although the UML model had different versions and the structure of the classes has been changed several times, this diagram reflects the final version of the model we're using on the software.



There are some things to comment about this diagram. The first, the __init__ method, which is the constructor for Python objects. In the case of the Species class, we also have the __eq__ method, which is used to compare the atomic composition of two different species (the comparison of other attributes of the Species is addressed further in this document, as it has its specific methods); the __hash__ function is needed to use the class Species as the key value for dictionaries.

The complexity of the diagram and the model itself comes when talking about the reactions, as the data that has to be stored there depends on some values, because the reactions can have different types attending to physical attributes. In our model, the class modelling the reactions is unique, thanks to the python flexibility when creating objects and its attributes, but on the diagram it is represented with different classes. As an example, if a reaction had its attribute isThreeBody=True, the object modelling this would be an instance of the Reaction class, but it also

would have the attributes corresponding to the ThreeBody class. Same will happen with TROE, Lindemann and Pressure Log reactions.

Another thing to point out is that in the case of the ThreeBody, TROE and Lindemann reactions, the attributes "reactants" and "products" will have a special key called "M" and its associated value will be a dictionary containing pairs [String:Float] representing the possible values for M.

There are some properties on the XML files that I was told to ignore, so they won't be modelled into the classes.

4.2. Reading data

As explained in the "Context" section, the info that we need to read is contained in two XML files, that follow a schema predefined. From these files, we need to extract the different species that appear on the model and the chemical reactions; all of them with its properties. In the case of the chemical reactions, this task is more complex as they are classified among different criteria, and each of the reaction types have different properties and values.

The first thing we will attend is the insertion of the species, as it is easier than the reactions. This subject is done by the function getSpecies (on the getSpecies.py file) and the auxiliar function parseSpecies (on the parseSpecies.py file), where the second one is called by the first one.

The parseSpecies function gets as parameter the file that must be read (the one that defines the kinetics of the model), and using the xml Python library, it extracts the elements that characterise the species: the name, its composition, and the kinetic parameters of the species, which are the NASA-Coefficients and the Lennard Jones Coefficients. All this information is returned into lists to the getSpecies function, and then iterates through the list of species to create one Species instance for each of the elements that were read. One thing that has to be pointed is that the getSpecies function gets only one parameter, which is the name of the file containing the kinetics model.

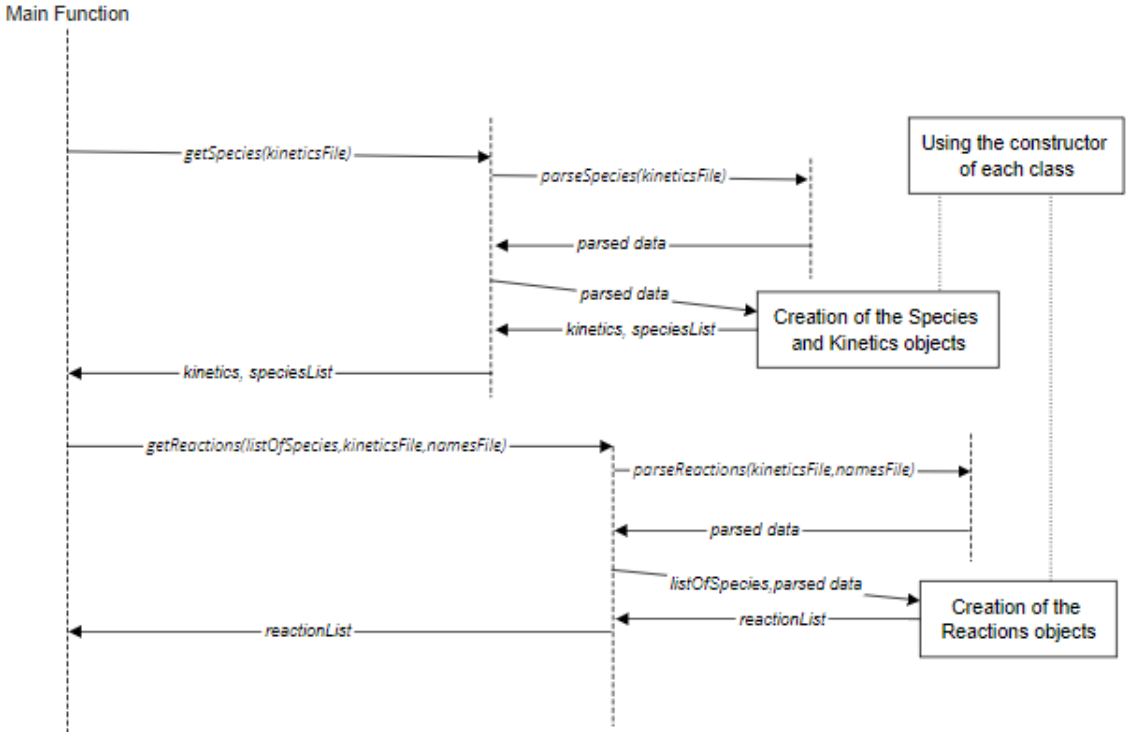
After the creation of all the instances of the Species class, then the Kinetics object is created, which contains the information related to the model. The acquisition of the data related to the Kinetics object is done inside the constructor of this class, as it is not a difficult task. The getSpecies function, which is called by the application to be able to interact with the model, returns a list of Species objects and the Kinetics object.

Now, we move on to read the files and get from them the information of the chemical reactions that are given in the model. On a parallel way as it happened with the species, we do that with getReactions function (contained on the file getReactions.py) and the auxiliar parseReactions function (on the parseReactions.py file).

The parseReactions function gets two parameters, which are the names of the two files that define the model: the kinetics file and the file containing all the names of all the chemical reactions. Again, this function will use the xml library and return a number of lists, the first containing the names of the reactions and then the different parameters that characterize the reactions, and the indexes of the reactions that have to be treated differently as they have specific properties, with their specific attributes.

The getReactions function gets three parameters. The first one is the list of the species that have been read from the model. Then, the names of the files that are needed to get the information. On this function the first action is to call the parseReaction function, and then create a list of chemical reactions object, which is returned to the user.

In this diagram we can see the sequence of calling all the functions explained above, that are needed to get the Python objects that represent the whole model. This diagram does not suit perfectly to the UML standard, but it is used as an auxiliar way to clarify the interactions on this part of the software.



4.3. Plotting species networks

The next step in the development of the project was the creation and visualization of the species network. The idea was to create an undirected node graph, where the nodes would be the species and there would be connections between them if they appeared in the same chemical reaction.

For this purpose, I used two different external libraries. The first one is NetworkX, a library that gives us methods to easily create nodes and edges, as well as an easy way to access properties of the network. The other library used is plotly, which provides us a way to plot graphs (a network graph in our case) which will open in the web navigator. This allows us to interact with the graph, what is really important in our case, and is a great advantage in relation to other libraries more frequently used like matplotlib.

All the methods related to the plotting of the network (and other tasks related to the network disambiguation) are contained in the file called networkFunctions.py. The user will call the method getNetwork, which gets as parameters the species list, the reaction list, and a list of the names of species which want to be seen on the graph. The reason behind the last parameter is that the model that I was given to train had more than 500 species and 20000 reactions, so plotting the whole

network graph was too time consuming due to the complexity. Because of that, the user will specify the species in which he is interested, and the network plotted will contain only the species with relations to those.

The already explained function uses two different methods, which are generateNetwork and plotNetwork. The first one gets the species and reactions lists and returns the network object, while the second one gets the species that the user wants to see and the network object and plots the result, opening it on the web browser.

4.4. Comparing species

As it was already explained before, the goal of the project was to be able to compare species, to be able to disambiguate them, telling if they are referring to the same component even if they have other names or the properties are differently expressed. The core of this functionality is on the file compare.py, but it uses other methods, as will be explained later.

The method to start the comparison between species is compareSpecies. It receives as parameters the name of the element which wants to be compared, a list of the elements against will be compared, two lists of species and two lists of reactions (one of each for each of the models being compared).

This function will use the generateNetwork function, which was explained before, and the getDistanceBetweenSpecies method, which gets as parameters two species and its two network graphs, in order to get different values of distance, related to the physical/chemical properties and the network distance.

All the values received from the different comparisons between species will be added to different arrays and plotted into bar graphs, using the library matplotlib to display the graph.

At this point, it is worth explaining which is the method that we used to determine the difference. We use three values, which will be consider as “coordinates”. The first one is the NASA-Coefficient distance. It is computed as the square root of the sum of difference between NASA-Coefficients values (ordered, so the lowest value in one species is compared with the lowest on the other) to the power of two. The same approach is used with the Lennard Jones coefficients, which is our second coordinate. Then these two values are then normalized, dividing them into the square root of the sum of the values raised to the power of two. The resulting values will be the ones returned and plotted on the graph.

The last coordinate of the distance is the value measured on the graph. To compute this, we use a function called getPropertiesOfNode (contained on networkFunctions.py file), which gets as parameters the graph object and the name of the species. This method returns for the species the centrality, betweenness, load centrality, closeness centrality and eigenvector centrality. With this values for both species, we get the square root of the sum of the power of the differences between each of the properties; which is our third coordinate. The absolute distance is then computed, as the norm of a three dimension vector would be.

4.5. Other functions implemented

Aside from the functions that were specified before and aimed to work in the topic of the entity disambiguation, I was also requested to do other tasks.

The first one was to plot the evolution of the forward reaction rate of the different chemical reactions, depending on temperature and pressure. This is done with mathematical formulas already predefined, and the only task was to implement them. These formulas depend on the type of the reaction, so we have to implement different functions for the elementary, the threebody, the TROE, the lindemann and the pressure log reactions. For some of them, the dependence was only on the temperature of the system, and others depended also on the pressure.

The first of the implemented functions is the corresponding to the elementary and threebody reactions (it is the same formula for both types) which is only dependant on the temperature. This is implemented in the function plotKElementary, which receives as parameters the reaction, the Tmin and Tmax; and plots a logarithmic graph where the forward reaction rate is drawn against the inverse of the temperature. The formula for this two types of reactions is the following:

$$K(T)=A*T^{b}*e^{-E/(RT)}$$

Where A is the lnA value, b is the Beta value and E/R is the eOverR value, all of them extracted from the kinetics file and included in the objects representing the chemical reactions.